Finite Automata

In this section we start our formal investigations into various models of computation, starting with one of the simplest forms: finite automata.

Reading

Sections 1.1

Practice problems (page 83): 1.1, 1.3, 1.4, 1.6, 1.12, 1.36, 1.37

Finite Automata

Examples and State Diagrams

The idea of a finite automaton is simple:

- You have some number of "states" that you can be in.
- There is some state you start from.
- Then on each input there's a specific state you go to. And you forget everything about what happened before.
- When you run out of input, you look at the state you are in, and see if it is one of the select few states that would be considered "final".

We usually represent finite automata via a **state diagram**:

> In the *state diagram* for a finite automaton, you have:
>
> - One "circle" for each state.
> - For each state arrows going out to other states, with a label on the arrows indicating the inputs for which they are applicable.
> - An arrow coming from nowhere indicates the *start state*.
> - An extra circle around a state indicates it is an *accept state*.

As an example, let us think of inputs being binary numbers (only 0 and 1 as digits), and we want a finite automaton that detects if the number is divisible by 3. Recall that a number is divisible by 3 if and only if the sum of its digits are. Since we see the digits one at a time, we will keep track of the remainder on division by 3 at each stage. So:

- We will have 3 states, one for remainder 0, one for remainder 1 and one for remainder 2.

- The 0 state is the start state.
- The 0 state is also the only accept state.
- From each state we have an arrow to itself, for input 0.
- For input 1 we have arrows from state 0 to state 1, from state 1 to state 2 and from state 2 to state 0.

Now as practice for you: Suppose we want a finite automaton that detects division by 7. We can still do it by keeping track of the remainder, but some changes would need to be made:

- We now have 7 states, corresponding to the remainders 0 through 6.
- If we are at a state k, meaning the remainder of our current number when divided by 7 is k, then we need to figure out what happens when we add one more digit. This is easy to do because of the properties of modular arithmetic, and because we can describe the effect of adding one more digit to the number: The number is doubled, and increased by the digit we are adding.
- So for example if we are at state 3, meaning the remainder is now 3, then on an input 0 we got to state 6 and on input 1 we go to state 0.
- Similarly if we are at state 5, then because $2 \times 5 = 3 \mod 7$ we would be going to state 3 on input 0 and to state 4 on input 1.

Exercise 1.37 asks you to generalize this idea.

Another practice problem: Describe an automaton that takes as input a string of a's and b's, and accepts only those strings where no letter appears twice in a row.

Formal Definitions

Eventually we want to have clear unambiguous definitions, so here is a more precise definition of finite automata:

A **(Deterministic) Finite Automaton** (DFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a finite set, called the *states*,
- Σ is a finite set, called the *alphabet*,
- There is a function $\delta\colon Q \times \Sigma \to Q$ called the *transition function*,
- There is a special state, $q_0 \in Q$ called the *start state*, (in particular Q must be nonempty)
- $F \subset Q$ is the set of *accept or final states* (possibly empty)

Question 1: What happens if F is the empty set? What if $F = S$?

Question 2: Use this formal definition to describe the automaton for division by 3 that we described earlier.

Question 3: Do the same for the automaton for division by 2.

We now discuss the meaning of computation in this setting. Informally we compute as follows:

- We start at the start state.
- We also have a string from Σ. We want to see if the DFA will accept it.
- We go through each letter in the string in order, and use the δ function to transition to a new state each time.
- When we run out of input, we see if we are at an accept/final state. If so then we "accept" the string.

More formally:

Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA.

- If $w = w_1 w_2 \cdots w_n$ is a string, where each $w_i \in \Sigma$, then we say that the DFA M **accepts** the string w if there is a sequence of states $r_0, \dots, r_n$ such that:
  - $r_0 = q_0$ is the start state
  - $\delta(r_i, w_{i+1}) = r_{i+1}$, i.e. each new input moves us to the next state
  - $r_n \in F$, i.e. the last state is in the special subset of accept states.
- We say that the DFA M **recognizes** the language L, and we write $L(M) = L$, if M accepts exactly those strings that are in L.
- A language is called a **regular language** if there is a DFA that recognizes it.

Note that there might be many DFAs all recognizing the same language. But for a given DFA there is exactly one language it recognizes, namely the language of all strings in Σ that the DFA accepts.

Practice problem: Construct DFAs (including writing out the formal definition) for the following languages:

- The language containing no strings at all.
- The language containing all strings.
- The language containing exactly the empty string.

- The language containing exactly one string, namely "a", where 'a' is a specific letter in the alphabet.
- The language containing all strings "", "a", "aa", "aaa", and so on (including the empty string).
- The language containing all strings "a", "aa", "aaa", and so on (excluding the empty string).

Practice problem: If we have the DFA for a language, what would the DFA for the "complement" language be?

The Union of Regular Languages

Regularity of a language (i.e. having a DFA that recognizes it) is preserved under a number of operations. In this section we will focus on the union:

> **Theorem**
> If A_1 and A_2 are two regular languages, then their union $A_1 \cup A_2$ is also regular.

In this section we will prove this theorem.

- We start with assuming that $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ are DFAs that recognize the languages A_1 and A_2 respectively (these exist since those languages are regular).
- We will construct a new DFA, M, which recognizes the union $A_1 \cup A_2$.
- The idea is this: M will "simulate" following both M_1 and M_2. At any stage we try to make progress towards both automata, and at the end we'll be able to see if one of the automata can accept.
- Now we proceed with the details. For a new automaton we need to define a 5-tuple: $(Q, \Sigma, \delta, q, F)$
  - Q will be the cartesian product $Q_1 \times Q_2$. In other words a state in the new automaton is a pair (q, q'), where $q \in Q_1$ and $q' \in Q_2$ are states from the two DFAs respectively (Some states in Q might end up being not reachable, but we don't care about that).
  - The transition function δ uses δ_1 and δ_2:

$$\delta\left((q, q'), a\right) = \left(\delta_1(q, a), \delta_2(q', a)\right)$$

  - The start state is the pair of start states, (q_1, q_2).

- The final states F are those that are final for M_1 or M_2:

$$F = \{(q, q') \mid q \in F_1 \text{ or } q' \in F_2\}$$

Question: What do you think about the intersection of regular languages?

The concatenation of regular languages, as well as the kleene star of a regular language, are regular, but this fact will be harder to prove, and it will be a consequence of further work we will do with non-deterministic automata and regular expressions.